

Department I - C Plus Plus

Modern and Lucid C++  
for Professional Programmers

Week 5 – Classes and Operators

Thomas Corbat / Felix Morgner  
Rapperswil, 13.10.2020  
HS2020



INSTITUTE FOR  
SOFTWARE

- **You can implement your own data types**
- **You know the elements a class consists of**
- **You know how to overload operators for classes**
- **You know the correct way to read and print objects**

- **Recap Week 4**

- **Classes**

- Declaration / Implementation
- Access Specifiers
- Constructors
- Inheritance

- **Operators**

- Members
- Free Operators

## Recap Week 4



- What would be the correct signature choice for printDocument, if the type of document

- ... is copyable
- ... potentially huge
- ... not modified in printDocument

```
void printDocument(<Type> document) {  
    for (auto const & line : document.content()) {  
        printLine(line);  
    }  
}
```

- Possibilities

- void printDocument(Document document)
- void printDocument(Document const document)
- void printDocument(Document & document)
- void printDocument(Document const & document)

- What is wrong with the following test case, that should check whether an `std::out_of_range` exception is thrown upon accessing an empty vector?

```
void testForExceptionTryCatch() {  
    std::vector<int> empty_vector{};  
    try {  
        empty_vector.at(0);  
    } catch (std::out_of_range const & e) {  
    }  
}
```

# Classes



## Goals:

- You know how to define a class in C++
- You know the elements a class consists of
- You can implement your own data types

- **Does one thing well and is named after that**
  - High Cohesion
- **Consists of member functions with only a few lines**
  - Avoid deeply nested control structures
- **Has a class invariant**
  - Provides a guarantee about its state (values of the member variables)
  - Constructors establish that invariant
- **Is easy to use without complicated protocol sequence requirements**

GoodClassName.h

```
class <GoodClassName> {  
  
    <member variables>  
  
    <constructors>  
  
    <member functions>  
};
```



- A class defines a new type
- A class is usually defined in a header file
- At the end of a class definition a semicolon is required

```

Date.h
#ifndef DATE_H_
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year{year}, month{month}, day{day} { /*...*/ }

    static bool isLeapYear(int year) { /*...*/ }

private:
    bool isValidDate() const { /*...*/ }
};

#endif /* DATE_H_ */
```

- **Include guard ensures that the content of a header file is only included once**

- Eliminates cyclic dependencies of `#include` directives

- **Prevents violation of the one definition rule**

- **Directives**

- `#ifndef <name>`
- `#define <name>`
- `#endif`

```
#ifndef DATE_H_                                     Date.h
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year{year}, month{month}, day{day} { /*...*/ }

    static bool isLeapYear(int year) { /*...*/ }

private:
    bool isValidDate() const { /*...*/ }
};

#endif /* DATE_H_ */
```

- **Keywords for defining a class**

- class
- struct

- **Default visibility for members of the class are**

- private for class
- public for struct

```
struct <name> {  
    ...  
};
```

```
#ifndef DATE_H_ Date.h  
#define DATE_H_  
  
class Date {  
    int year, month, day;  
public:  
    Date(int year, int month, int day)  
        : year{year}, month{month}, day{day} { /*...*/}  
  
    static bool isLeapYear(int year) { /*...*/}  
  
private:  
    bool isValidDate() const { /*...*/}  
};  
  
#endif /* DATE_H_ */
```

- **Access specifiers (followed by a colon :)**
  - **private:**  
visible only inside the class (and friends); for hidden data members
  - **protected:**  
also visible in subclasses
  - **public:**  
visible from everywhere; for the interface of the class
- **All subsequent members have this visibility**
- **Each visibility can reoccur multiple times**

```
#ifndef DATE_H_ Date.h
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year{year}, month{month}, day{day} { /*...*/ }

    static bool isLeapYear(int year) { /*...*/ }

private:
    bool isValidDate() const { /*...*/ }
};

#endif /* DATE_H_ */
```

- Have a type and a name

`<type> <name>;`

- The content/state of an object that represents the value of the type
- Don't make member variables **const** as it prevents copy assignment
- Don't add members to communicate between member function calls
  - Hard to test
  - Such usage protocols are a burden for the user of the class
  - Better: Use parameters

```
#ifndef DATE_H_                                     Date.h
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year{year}, month{month}, day{day} { /*...*/ }

    static bool isLeapYear(int year) { /*...*/ }

private:
    bool isValidDate() const { /*...*/ }
};

#endif /* DATE_H_ */
```

- **Function with name of the class**

- Special member function

- **No return type**

```
<class name>(){}  

```

- **Initializer list for member initialization**

```
<class name>(<parameters>)  
: <initializer-list>  
{}
```

```
#ifndef DATE_H_ Date.h  
#define DATE_H_  
  
class Date {  
    int year, month, day;  
public:  
    Date(int year, int month, int day)  
        : year{year}, month{month}, day{day} { /*...*/ }  
  
    static bool isLeapYear(int year) { /*...*/ }  
  
private:  
    bool isValidDate() const { /*...*/ }  
};  
  
#endif /* DATE_H_ */  

```

- **Default Constructor**

Date d{};

- No parameters
- Implicitly available if there are no other explicit constructors
- Has to initialize member variables with default values

- **Copy Constructor**

Date d2{d};

- Has one <own-type> const & parameter
- Implicitly available (unless there is an explicit move constructor or assignment operator)
- Copies all member variables
- Usually you don't need to implement it explicitly

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Typeconversion-Constructor  
    explicit Date(std::string const &);  
    //Destructor  
    ~Date();  
};
```



- **Move Constructor**

```
Date d2{std::move(d)};
```

- Has one <own-type> && parameter
  - Implicitly available (unless there is an explicit copy constructor or assignment operator)
  - Moves all members
  - Usually you don't need to implement it explicitly
- **Will be covered in C++ Advanced**

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Typeconversion-Constructor  
    explicit Date(std::string const &);  
    //Destructor  
    ~Date();  
};
```



- **Typeconversion Constructor**

```
Date tomorrow{"19/10/2017"s};
```

- Has one <other-type> const & parameter
- Converts the input type if possible
- Declare **explicit** to avoid unexpected conversions!

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Typeconversion-Constructor  
    explicit Date(std::string const &);  
    //Destructor  
    ~Date();  
};
```

- **Initializer List Constructor**

```
Container box{item1, item2, item3};
```

- Has one `std::initializer_list` parameter
- Does not need to be **explicit**, implicit conversion is usually desired

```
struct Container {  
    Ccontainer() = default;  
    Container(std::initializer_list<Element> elements);  
private:  
    std::vector<Element> elements{};  
};
```

- **Initializer List constructors are preferred if a variable is initialized with {}**

```
std::vector v(5, 10)
```

```
std::vector v{5, 10}
```

- **Named like the default constructor but with a leading ~**

```
~Date();
```

- **Must release all resources**
- **Implicitly available**
  - If you program properly you will hardly ever need to implement it yourself!
- **Must not throw an exception!**
- **Called automatically at the end of the block for local instances**

```
class Date {  
public:  
    Date(int year, int month, int day);  
    //Default-Constructor  
    Date();  
    //Copy-Constructor  
    Date(Date const &);  
    //Move-Constructor  
    Date(Date &&);  
    //Typeconversion-Constructor  
    explicit Date(std::string const &);  
    //Destructor  
    ~Date();  
};
```

- Base classes are specified after the name

```
class <name> : <base1>, ..., <baseN>
```

- Multiple inheritance is possible
- Inheritance can specify a visibility
  - public, protected, private
  - Limits the **maximum** visibility of the inherited members
  - If no visibility is specified the default of the inheriting class is used (class->private and struct->public)
- Details about Diamonds and Virtual inheritance later

```
class Base {  
private:  
    int onlyInBase;  
protected:  
    int baseAndInSubclasses;  
public:  
    int everyoneCanFiddleWithMe  
};
```

```
class Sub : public Base {  
    //Can see baseAndInSubclasses and  
    //everyoneCanFiddleWithMe  
};
```

```
#ifndef DATE_H_                                     Date.h
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static bool isLeapYear(int year);

private:
    bool isValidDate() const;
};

#endif /* DATE_H_ */
```

```
#include "Date.h"                                     Date.cpp

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    /*...*/
}

bool Date::isLeapYear(int year) {
    /*...*/
}

bool Date::isValidDate() const {
    /*...*/
}
```



```
#include "Date.h"                                     Any.cpp

void foo() {
    Date today{2016, 10, 19};

    auto thursday{today.tomorrow()};

    Date::isLeapYear(2016);

    //what now?
    Date invalidDate {2016, 13, 1};
}
```

```
#ifndef DATE_H_                                       Date.h
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    Date tomorrow() const;

    static bool isLeapYear(int year);

private:
    bool isValidDate() const;
};

#endif /* DATE_H_ */
```

```
struct Recipe {  
    Recipe(std::initializer_list<Step> steps);  
    Meal cook() const;  
private:  
    std::vector<Step> steps{};  
};
```

Correct

The class declaration has a semicolon at the end and a Recipe is constructible with a list of (cooking) steps. The visibilities are sensible.

```
struct Vehicle {  
    Location location{};  
};  
class Car : Vehicle {  
public:  
    Route drive(Destination destination);  
};  
void printLocation(Car & car) {  
    std::cout << car.location;  
}
```

Inncorrect

While Vehicle and Car are syntactically correct, Car inherits privately from Vehicle (due to the class keyword). Therefore, the members of Vehicle cannot be accessed from outside the Car class.

- **Establish Invariant**

- Properties for a value of the type that are always true
- E.g. a Date instance always represents a valid date
- All (public) member functions assume and keep it intact

- **Initialize all members**

- Constructors only create a valid instance (Otherwise throw an exception!)
- Use initializer lists
- Use default values if possible/necessary

Date.cpp

```
#include "Date.h"

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    if (!isValidDate()) {
        throw std::out_of_range{"invalid date"};
    }
}

Date::Date() : Date{1980, 1, 1} {
}

Date::Date(Date const & other)
    : Date{other.year, other.month, other.day} {
}
```



- **As we have specified a default value, this should be the value created by the default constructor**

- Constructor without parameters

- **Remark regarding initialization:**

```
Date nice_d{};  
Date ugly_d;
```

- Both create a Date and call the default constructor in this case
- The second (without {}) does not work with all types. It might contain uninitialized variables
- Good practice: Initialize all variables with {}!

```
#ifndef DATE_H_                                     Date.h  
#define DATE_H_  
  
class Date {  
    //...  
    Date();  
};  
  
#endif /* DATE_H_ */
```

```
#include "Date.h"                                     Date.cpp  
  
Date::Date()  
    : year{9999}, month{12}, day{31} {  
}
```

- **Member variables can have a default value assigned**
  - NSDMI – Non-Static Data Member Initializers

```
class <classname> {  
    <type> <membername>{<default-value>};  
};
```

- **Such values are used if the member is not present in the initializer list of the constructor**
  - Initializer list still overrides those values
- **Useful if multiple constructors initialize data similarly**
  - Avoids duplication

```
#ifndef DATE_H_                                     Date.h  
#define DATE_H_  
  
class Date {  
    int year{9999}, month{12}, day{31};  
    //...  
    Date();  
    Date(int year, int month, int day);  
};  
  
#endif /* DATE_H_ */
```

```
#include "Date.h"                                     Date.cpp  
  
Date::Date() {  
}  
  
Date::Date(int year, int month, int day)  
    : year{year}, month{month}, day{day} {  
    /*...*/  
}
```

- **Some special member functions are implicitly available in certain cases**
  - E.g. Default constructor is implicitly available if no other explicit constructor is declared
- **(Re-)implementing the default behavior of the default constructor can be avoided by defaulting it**

```
<ctor-name>() = default;
```

- Adds the corresponding constructor to the type with the same behavior as if it was implicitly available
- Possible for:
  - Default constructor and destructor
  - Copy/move constructor
  - Copy/move assignment operator

```
#ifndef DATE_H_                                     Date.h
#define DATE_H_

class Date {
    int year{9999}, month{12}, day{31};
    //...
    Date() = default;
    Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

```
#include "Date.h"                                     Date.cpp

Date::Date(){
}

Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
    /*...*/
}
```

- **Some special member functions are implicitly available in certain cases**
    - E.g. Default constructor is implicitly available if no other explicit constructor is declared
  - **Those implicit constructor are not always wanted**
    - Make them explicitly private
    - Or delete them
- `<ctor-name>() = delete;`
- Possible for:
    - Default constructor and destructor
    - Copy/move constructor
    - Copy/move assignment operator

```
Banknote.h
#ifndef BANKNOTE_H_
#define BANKNOTE_H_

class Banknote {
    int value;
    //...
    Banknote(Banknote & const) = delete;
};

#endif /* BANKNOTE_H_ */
```

```
Forger.cpp
#include "Banknote.h"

Banknote forge(Banknote const & note) {
    Banknote copy{note}; //! ⚡
    return copy;
}
```

- Similar to Java constructors can call other constructors

C++

```
Ctor(Parameters)  
  : Ctor(Arguments) {  
}
```

Java

```
Ctor(Parameters) {  
    this(Arguments);  
}
```

- Constructor call has to be in the member initializer list
- Similar are calls to constructors of base classes

C++


```
Ctor(Parameters)  
  : Base(Arguments) {  
}
```

Java

```
Ctor(Parameters) {  
    super(Arguments);  
}
```

```
#ifndef DATE_H_                                     Date.h  
#define DATE_H_  
  
class Date {  
    //...  
    Date(int year, Month month, int day);  
    Date(int year, int month, int day);  
};  
  
#endif /* DATE_H_ */
```

```
#include "Date.h"                                     Date.cpp  
  
Date::Date(int year, int month, int day)  
    : Date{year, Month(month), day} {}
```



- Don't violate invariant

- Leave object in valid state

- Implicit **this** object

- Is a pointer
- Member access with arrow: ->

- Declare **const** if possible!

- Must not modify members if **const**

- Refers to **this** object
- Can only call **const** members

- Otherwise access to all other members

```
#include "Date.h" Date.cpp

bool Date::isValidDate() const {
    if (day <= 0) {
        return false;
    }
    switch (month) {
        case 1: case 3: case 5: case 7:
        case 8: case 10: case 12:
            return day <= 31;
        case 4: case 6: case 9: case 11:
            return this->day <= 30;
        case 2:
            return day <= (isLeapYear(year) ? 29:28);
        default:
            return false;
    }
}
```

```
struct Counter {  
    void increase(unsigned step) {  
        auto before = current();  
        value = before + step;  
    }  
    unsigned current() const;  
private:  
    unsigned value;  
};
```

Correct

It is allowed to call const member functions from non-const member functions.

```
struct Document {  
    void print(std::ostream & out) const {  
        updatePrintDate();  
        out << content;  
    }  
private:  
    void updatePrintDate();  
};
```

Inncorrect

It is not allowed to call a non-const member function from a const member function

- No **this** object
- Cannot be **const**
- No **static** keyword in implementation
- Call with `<classname>::<member>()`

```
Date::isLeapYear(2016);
```

```
class Date {                                     Date.h
    //...
    static bool isLeapYear(int year);
};
```

```
#include "Date.h"                                Date.cpp

bool Date::isLeapYear(int year) {
    if (year % 400 == 0) {
        return true;
    } else if (year % 100 == 0) {
        return false;
    } else if (year % 4 == 0) {
        return true;
    }
    return false;
}

//or the unreadable version
bool Date::isLeapYear(int year) {
    return !(year % 4) &&
        ((year % 100) || !(year % 400));
}
```



- No **static** keyword in implementation
- **static const** member can be initialized directly
- Access outside class with name qualifier: `<classname>::<member>`

```
class Date {                                     Date.h
    static const Date myBirthday;
    static Date favoriteStudentsBirthday;
    static const Date today{2018, 10, 16};

    //...
};
```

```
#include "Date.h"                               Any.cpp
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
    Date::favoriteStudentsBirthday = ...;
}
```

```
#include "Date.h"                               Date.cpp

Date const Date::myBirthday{1964, 12, 24};
Date Date::favoriteStudentsBirthday{1995, 5, 10};
```

```
struct Recipe {  
    Recipe(std::vector<Step> steps) = default;  
    Meal cook() const;  
private:  
    std::vector<Step> steps{};  
};
```

Incorrect

Of all constructors, only default, copy and move constructors can be declared = `default`.

```
struct Chair {  
    explicit Chair(unsigned legs = 4u);  
private:  
    unsigned legs;  
};  
Chair::Chair(unsigned legs)  
    : legs{legs} {  
    if (legs < 1u) {  
        throw std::invalid_argument{"..."};  
    }  
}
```

Correct

Constructors can have default arguments too. In the declaration, a constructor that can be called with a single argument should be explicit. It uses the member initializer list and throws an exception if the invariant cannot be established.

# Operator Overloading



## Goals:

- You know how to overload operators for classes
- You know the correct way to read and print objects
- You can deal with streams correctly in your classes

- Custom operators can be overloaded for user-defined types
- Declared like a function, with a special name

```
<returntype> operator op(<parameters>);
```

- Unary operators -> one parameter
  - Binary operators -> two parameters
- Implement operators reasonably!
  - Semantic should be natural
- "When in doubt, do as the **ints** do"
  - Scott Meyers – Effective C++

- Overloadable Operators (*op*):

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- Non-Overloadable Operators:

```
::            .*            .            ?:
```

- **Example: Making Date comparable**
- **Compare year, month and day**
- **Free operator<**
  - Two parameters of type **Date**
  - Each **const** &
  - Return type **bool**
- **inline** when defined in header
- **Problem with free operator**
  - No access to private members

```
#include "Date.h"
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
    Date d{};
    std::cin >> d;
    std::cout << "is d older? " << (d < Date::myBirthday);
}
```

Any.cpp

```
class Date {
    int year, month, day; //private ☹️
};

inline bool operator<(Date const & lhs, Date const & rhs) {
    return lhs.year < rhs.year ||
        (lhs.year == rhs.year && (lhs.month < rhs.month ||
            (lhs.month == rhs.month && lhs.day == rhs.day)));
}
```

Date.h

- **Member operator<**

- One parameters of type **Date**
- Which is **const** &
- Return type **bool**
- Right-hand side of operation

- **Implicit **this** object**

- **const** due to qualifier
- Left-hand side of operation

- **Access to private members**

- **Implicit **inline** as member**

```
#include "Date.h"
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
    Date d{};
    std::cin >> d;
    std::cout << "is d older? " << (d < Date::myBirthday);
}
```

Any.cpp

```
class Date {
    int year, month, day; //private 😊

    bool operator<(Date const & rhs) const {
        return year < rhs.year ||
            (year == rhs.year && (month < rhs.month ||
                (month == rhs.month && day == rhs.day)));
    }
};
```

Date.h

- `std::tie` creates a tuple and binds the arguments with lvalue references
- `std::tuple` provides comparison operators:
  - `operator==`  
`operator!=`  
`operator<`  
`operator<=`  
`operator>`  
`operator>=`
- Comparison of `std::tuple` is component-wise from left to right

```
#include "Date.h"                                     Date.cpp
#include <tuple>

bool Date::operator<(Date const & rhs) const {
    return std::tie(year, month, day) <
           std::tie(rhs.year, rhs.month, rhs.day);
}
```

```
class Date {                                           Date.h
    int year, month, day; //private 😊

    bool operator<(Date const & rhs) const;
};
```

- **Ensure**
  - Transitivity
  - Associativity
  - Commutativity
- **Avoid duplication!**
- **Beware of call loops!**

```
class Date { Date.h  
    int year, month, day; //private 😊  
public:  
    bool operator<(Date const & rhs) const;  
};  
  
inline bool operator>(Date const & lhs, Date const & rhs) {  
    return rhs < lhs;  
}  
  
inline bool operator>=(Date const & lhs, Date const & rhs) {  
    return !(lhs < rhs);  
}  
  
inline bool operator<=(Date const & lhs, Date const & rhs) {  
    return !(rhs < lhs);  
}  
  
inline bool operator==(Date const & lhs, Date const & rhs) {  
    return !(lhs < rhs) && !(rhs < lhs);  
}  
  
inline bool operator!=(Date const & lhs, Date const & rhs) {  
    return !(lhs == rhs);  
}
```



- Boost provides base classes to implement (inherit) derived operators
- Inherit from `boost::less_than_comparable`
  - `private` inheritance is enough
- `boost::less_than_comparable`
  - requires <
  - provides >, <= and >=
- See boost documentation
  - [www.boost.org](http://www.boost.org)

```
#include "boost/operators.hpp"                                Date.h
#include <tuple>

class Date : private boost::less_than_comparable<Date> {
    int year, month, day;
public:
    bool operator<(Date const & rhs) const {
        return std::tie(year, month, day) <
               std::tie(rhs.year, rhs.month, rhs.day);
    }
};
```

- **Output operator as free operator:**

- `operator<<`
- Parameters: `std::ostream &` and `Date const &`
- Returns `std::ostream &` for chaining output

```
#include "Date.h"
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
}
```

Any.cpp

- **Problem: Cannot access private members!**

```
#include <ostream>

class Date {
    int year, month, day; //private ☹️
};

inline std::ostream & operator<<(std::ostream & os, Date const & date) {
    os << date.year << "/" << date.month << "/" << date.day; //Invalid ⚡
    return os;
}
```

Date.h

- **Output operator as member operator**

- `operator<<`
- Parameter: `std::ostream &`
- Can access private members
- Returns `std::ostream &` for chaining output


- **Problems**

- The compiler cannot accept the `std::ostream` on the left in calls
- `std::ostream` would be on the right-hand side, which is wrong by convention

```
#include "Date.h"
#include <iostream>
```

Any.cpp

```
void foo() {
    //Invalid (Compiler)
    std::cout << Date::myBirthday;
    //Invalid (by Convention)
    Date::myBirthday << std::cout;
}
```



```
#include <ostream>
```

Date.h

```
class Date {
    int year, month, day; //private 😊
    std::ostream & operator<<(std::ostream & os) const {
        os << year << "/" << month << "/" << day;
        return os;
    }
};
```

- **Workaround print member function**

- Has access to private data members
- Can be called from free operator<<

```
#include "Date.h"
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
}
```

Any.cpp

```
#include <ostream>

class Date {
    int year, month, day;
public:
    std::ostream & print(std::ostream & os) const {
        os << year << "/" << month << "/" << day;
        return os;
    }
};

inline std::ostream & operator<<(std::ostream & os, Date const & date) {
    return date.print(os);
}
```

Date.h

- **Input operator has the same problems as the output operator**

- `operator>>`
- Parameters: `std::istream &` and `Date &`
- Returns `std::istream &` for chaining input

```
#include "Date.h"      Any.cpp
#include <iostream>

void foo() {
    Date d{};
    std::cin >> d;
}
```

```
#include <iostream>      Date.h

class Date {
    int year, month, day;
public:
    std::istream & read(std::istream & is) {
        //Logic for reading values and verifying correctness
        return is;
    }
};

inline std::istream & operator>>(std::istream & is, Date & date) {
    return date.read(is);
}
```

- Expect `std::istream` to be in `good()` state as precondition and to provide a correct date
- If extracting a date fails set the `std::istream` to fail state
- Do not overwrite the `this` object if the input cannot be used to read a valid date object
  - Keep the invariant "Date represents a valid date"

```
//Includes Date.h  
  
class Date {  
    int year, month, day;  
public:  
    std::istream & read(std::istream & is) {  
        int year{-1}, month{-1}, day{-1};  
        char sep1, sep2;  
        //read values  
        is >> year >> sep1 >> month >> sep2 >> day;  
        try {  
            Date input{year, month, day};  
            //overwrite content of this object (copy-ctor)  
            (*this) = input;  
            //clear stream if read was ok  
            is.clear();  
        } catch (std::out_of_range const & e) {  
            //set failbit  
            is.setstate(std::ios::failbit);  
        }  
        return is;  
    }  
};
```

- Declaration of a constructor that takes an std::istream & as parameter

```
explicit Date(std::istream & in);
```

- Declare constructors with one parameter explicit to avoid automatic conversion
- Throw an exception if the input does not represent a valid date
  - For not violating the invariant
  - Alternative: Create a date with a default value

```
#ifndef DATE_H_
#define DATE_H_

class Date {
    //...
    explicit Date(std::istream & in);
};

#endif /* DATE_H_ */
```

Date.h

```
#include "Date.h"

Date::Date(std::istream & in)
    : year{}, month{}, day{} {
    read(in);
    if (in.fail())
        throw std::out_of_range{"invalid date"};
}
```

Date.cpp

- **Declaration of a factory function for Date**

```
Date make_date(std::istream & in);
```

- **Factory functions**

- make\_xxx() or create\_xxx()

- **Placed in**

- Class as static member function
- Or in same namespace as the class

- **Delivers a default value if reading fails**

- E.g. Date{9999, 12, 31}
- Similar to std::string::npos for "not found"

```
Date make_date(std::istream & in)
try {
    return Date{in};
} catch (std::out_of_range const &) {
    return Date{9999, 12, 31};
}
```



```
struct SwissGrid {
    SwissGrid() = default;
    SwissGrid(double y, double x);
    void read(std::istream & in) {
        double y{}; in >> y;
        double x{}; in >> x;
        try {
            SwissGrid inputCoordinate{y, x};
            *this = inputCoordinate;
        } catch(std::invalid_argument const & e) {
            in.setstate(std::ios_base::failbit);
        }
    }
private:
    double y{600000.0};
    double x{200000.0};
};

std::istream & operator>>(
    std::istream & in,
    SwissGrid & coordinate) {
    coordinate.read(in);
    return in;
}
```

Correct

The input (and output) operator must be implemented as free function. Since they usually won't have access to the private members of a type, a member function for reading/writing is required.

- **Input (>>) and Output (<<) as friend operator**

```
class Date {
    int year, month, day;
public:
    friend std::istream & operator>>(std::istream & is, Date & date);
    friend std::ostream & operator<<(std::ostream & os, Date const & date);
};
```

Date.h

```
#include "Date.h"

std::istream & operator>>(std::istream & is, Date & date) {
    //read logic
    return is;
}

std::ostream & operator<<(std::ostream & os, Date const & date) {
    //print logic
    return os;
}
```

Date.cpp

- **Separate the class declaration from the member function implementations properly into header and source files**
- **Initialize member variables with default values or in the constructor's initializer list**
- **Throw an exception from a constructor if it cannot establish the class invariant**
- **You need to implement input and output operators as free functions or as friends**
- **Provide sensible operations when implementing operators for your types**